Dynamic programming

Longest Common Subsequence

Algorithmic Paradigms

- Divide-and-conquer: Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems. i.e. (general algorithm design technique for solving problems defined by or formulated as recurrences with overlapping subinstances)

Dynamic Programming

Idea:

- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
- solve smaller instances once
- record solutions in a table
- extract solution to the initial instance from that table

Richard Bellman: Pioneered the systematic study of dynamic programming in the 1950s to solve optimization problems and later assimilated by CS

Dynamic Programming Applications

Areas:

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, systems, ….

Elements of DP

- Optimal (sub)structure
  - An optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping subproblems
  - The space of subproblems is “small” in that a recursive algorithm for the problem solves the same subproblems over and over. Total number of distinct subproblems is typically polynomial in input size.
- (Reconstruction an optimal solution)

Finding Optimal substructures

- Show a solution to the problem consists of making a choice, which results in one or more subproblems to be solved.

- Suppose you are given a choice leading to an optimal solution.
  – Determine which subproblems follows and how to characterize the resulting space of subproblems.

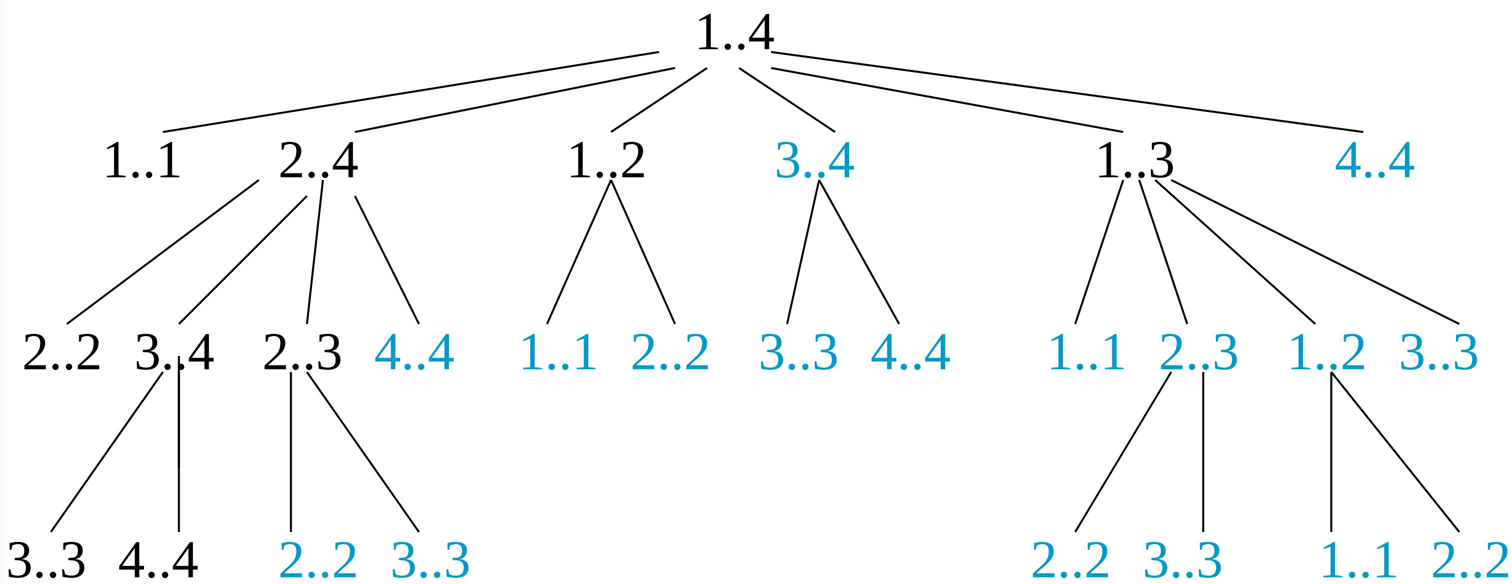
- Show the solution to the subproblems used within the optimal solution to the problem must themselves be optimal by cut-and-paste technique.

A Recursive Algorithm for Matrix-Chain Multiplication

RECURSIVE-MATRIX-CHAIN(p,i,j) (called with($p,1,n$))

1. **if** $i=j$ **then return** 0
2. $m[i,j] \leftarrow \infty$
3. **for** $k \leftarrow i$ to $j-1$
4. **do** $q \leftarrow$ RECURSIVE-MATRIX-CHAIN(p,i,k)+ RECURSIVE-MATRIX-CHAIN($p,k+1,j$)+$p_{i-1}p_kp_j$
5. **if** $q < m[i,j]$ **then** $m[i,j] \leftarrow q$
6. **return** $m[i,j]$

Recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p,1,4)



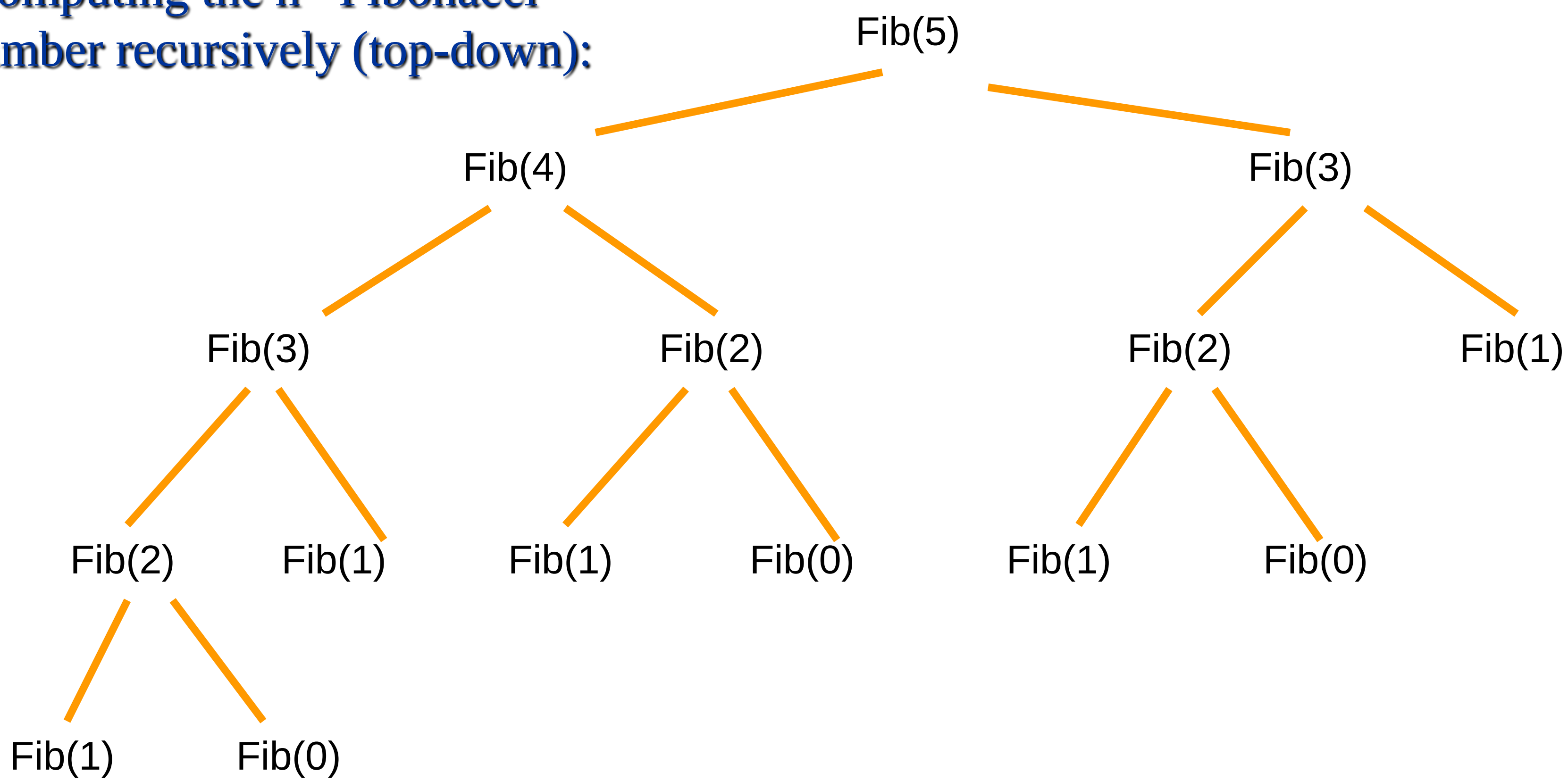
Drawback of Divide & Conquer

- Sometimes can be inefficient
- **Fibonacci numbers:**
  - $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for $n > 1$
- Sequence is 0, 1, 1, 2, 3, 5, 8, 13, …
- Obvious recursive algorithm:
- Fib(n):
  - if n = 0 or 1 then return n
  - else return (F(n-1) + Fib(n-2))

Computing Fibonacci Numbers

Recursion Tree for Fib(5)

Computing the n[th] Fibonacci number recursively (top-down):



How Many Recursive Calls?

- If all leaves had the same depth, then there would be about 2^n recursive calls.
- But this is over-counting.
- Exponential!

Save Work

- Wasteful approach - repeat work unnecessarily
  - Fib(2) is computed three times
- Recursion adds overhead
  - extra time for function calls
  - extra space to store information on the runtime stack about each currently active function call
- Avoid the recursion overhead by filling in the table entries bottom up, instead of top down.
  - Instead, compute Fib(2) once, store result in a table, and access it when needed

Dynamic Programming for Fibonacci

- Fib(n):
- F[0] := 0; F[1] := 1;
- for i := 2 to n do
    - F[i] := F[i-1] + F[i-2]
  - return F[*n*]

time reduced from exponential to linear!

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory for later use
- More efficient than "*brute-force methods*", which solve the same subproblems over and over again

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- if $|X| = m$, $|Y| = n$, then there are 2^m subsequences of x; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is $O(n\ 2^m)$
- Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.
- Subproblems: "find LCS of pairs of *prefixes* of X and Y"

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. $c[0,0] = 0$)
- LCS of empty string and any other string is empty, so for every i and j: $c[0, j] = c[i,0] = 0$

LCS recursive solution

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS recursive solution

- **Second case:** *x[i] != y[j]*

As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

LCS Length Algorithm

LCS-Length(X, Y)

1. m = length(X) // get the # of symbols in X
2. n = length(Y) // get the # of symbols in Y
3. for i = 1 to m   c[i,0] = 0   // special case: Y_0
4. for j = 1 to n   c[0,j] = 0   // special case: X_0
5. for i = 1 to m   // for all X_i
6. for j = 1 to n   // for all Y_j
7.   if (X_i == Y_j)
8.   c[i,j] = c[i-1,j-1] + 1
9.   else c[i,j] = max(c[i-1,j], c[i,j-1])
10. return c

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Example (0)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

for i = 1 to m c[i,0] = 0
for j = 1 to n c[0,j] = 0

LCS Example (2)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (3)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1** →	**1**
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (6)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

if (X_i == Y_j)
    c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj	B	D	C	A	B	
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	
3	C		0					
4	B		0					

if ($X_i == Y_j$)

$c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (8)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**					
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (10)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**			
4	**B**		**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (11)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**		
4	**B**	**0**					

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (12)

ABCB
BDCAB

		j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**	
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**	
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**	
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**	
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**	
4	**B**	**0**						

if (X_i == Y_j)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (13)

ABCB
BDCAB

		j	0	1	2	3	4	5
i			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**				

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	

if ($X_i == Y_j$)
$c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

LCS Example (15)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each *c[i,j]* depends on *c[i-1,j]* and *c[i,j-1]*

or *c[i-1, j-1]*

For each c[i,j] we can say how it was acquired:

2	2
2	3

For example, here

c[i,j] = c[i-1,j-1] +1 = 2+1=3

How to find actual LCS - continued

- Remember that

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- So we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**	**1**	**1**
2	**B**	**0**	**1**	**1**	**1**	**1**	**2**
3	**C**	**0**	**1**	**1**	**2**	**2**	**2**
4	**B**	**0**	**1**	**1**	**2**	**2**	**3**

Finding LCS (2)

i		j	0	1	2	3	4	5
		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1** ←	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2** ←	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	**3**

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)

Matrix-chain multiplication (MCM) -DP

- Problem: given <A_1, A_2, …,A_n>, compute the product: $A_1 \times A_2 \times \ldots \times A_n$, find the fastest way (i.e., minimum number of multiplications) to compute it.

- Suppose two matrices A(p,q) and B(q,r), compute their product C(p,r) in $p \times q \times r$ multiplications
  - **for** i=1 **to** p **for** j=1 **to** r $C[i,j]=0$
  - **for** i=1 **to** p
    - **for** j=1 **to** r
      - **for** k=1 **to** q $C[i,j] = C[i,j]+ A[i,k]B[k,j]$

Matrix-chain multiplication -DP

- Different parenthesizations will have different number of multiplications for product of multiple matrices
- Example: A(10,100), B(100,5), C(5,50)
  - If ((A $\times$B) $\times$C), 10 $\times$100 $\times$5 +10 $\times$5 $\times$50 =7500
  - If (A $\times$(B $\times$C)), 10 $\times$100 $\times$50+100 $\times$5 $\times$50=75000
- The first way is ten times faster than the second !!!
- Denote $<A_1, A_2, \ldots, A_n>$ by $< p_0, p_1, p_2, \ldots, p_n>$
  - i.e, $A_1(p_0,p_1)$, $A_2(p_1,p_2)$, …, $A_i(p_{i-1},p_i)$,… $A_n(p_{n-1},p_n)$

Matrix-chain multiplication –MCM DP

- Intuitive brute-force solution: Counting the number of parenthesizations by exhaustively checking all possible parenthesizations.

- Let P(n) denote the number of alternative parenthesizations of a sequence of n matrices:
  – $P(n) = \begin{cases} 1 & \text{if } n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$

- The solution to the recursion is $\Omega(2^n)$.
- So brute-force will not work.

Matrix-chain Multiplication

- $$C = A_1 A_2 \ldots A_n$$
- Different ways to compute C
  - $C = (A_1(A_2A_3)(A_4A_5))A_6$
  - $C = (A_1A_2)((A_3A_4)(A_5A_6))$
- Matrix multiplication is associative
  - So output will be the same
- However, time cost can be very different

MCP DP Steps

- Step 1: structure of an optimal parenthesization
  - Let $A_{i..j}$ ($i \leq j$) denote the matrix resulting from $A_i \times A_{i+1} \times \ldots \times A_j$
  - Any parenthesization of $A_i \times A_{i+1} \times \ldots \times A_j$ must split the product between A_k and A_{k+1} for some k, ($i \leq k < j$). The cost = # of computing $A_{i..k}$ + # of computing $A_{k+1..j}$ + # $A_{i..k} \times A_{k+1..j}$.
  - If k is the position for an optimal parenthesization, the parenthesization of "prefix" subchain $A_i \times A_{i+1} \times \ldots \times A_k$ within this optimal parenthesization of $A_i \times A_{i+1} \times \ldots \times A_j$ must be an optimal parenthesization of $A_i \times A_{i+1} \times \ldots \times A_k$.
  - $A_i \times A_{i+1} \times \ldots \times A_k \times A_{k+1} \times \ldots \times A_j$

MCP DP Steps

- Step 2: a recursive relation
  - Let m[i,j] be the minimum number of multiplications for $A_i \times A_{i+1} \times \ldots \times A_j$
  - m[1,n] will be the answer
  - $$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i<j \end{cases}$$

MCM DP Steps

- Step 3, Computing the optimal cost
  - If by recursive algorithm, exponential time $\Omega(2^n)$ (ref. to P.346 for the proof.), no better than brute-force.
  - Total number of subproblems: $\Theta(n^2)$
  - Recursive algorithm will encounter the same subproblem many times.
  - If tabling the answers for subproblems, each subproblem is only solved once.
  - The second hallmark of DP: overlapping subproblems and solve every subproblem just once.

MCM DP Steps

- Step 3, Algorithm,
  - array $m[1..n,1..n]$, with $m[i,j]$ records the optimal cost for $A_i \times A_{i+1} \times \ldots \times A_j$.
  - array $s[1..n,1..n]$, $s[i,j]$ records index k which achieved the optimal cost when computing $m[i,j]$.
  - Suppose the input to the algorithm is $p=< p_0 , p_1 ,\ldots, p_n >$.

MCM DP Steps

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
      do m[i, i] ← 0
for l ← 2 to n                    ▷ l is the chain length.
      do for i ← 1 to n − l + 1
              do j ← i + l − 1
                 m[i, j] ← ∞
                 for k ← i to j − 1
                      do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
                         if q < m[i, j]
                           then m[i, j] ← q
                                s[i, j] ← k
return m and s
```

MCM DP Example

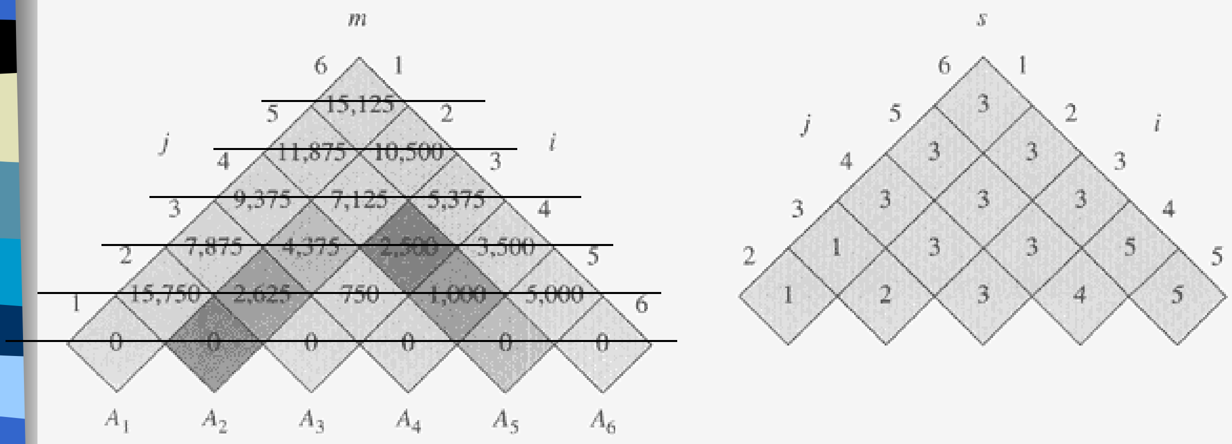


Figure 15.3 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

The tables are rotated so that the main diagonal runs horizontally. Only the main diagonal and upper triangle are used in the m table, and only the upper triangle is used in the s table. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 9 when computing

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 & = 13000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 & = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 & = 11375 \end{cases}$$
$$= 7125\,.$$

MCM DP Steps

- Step 4, constructing a parenthesization order for the optimal solution.
  - Since $s[1..n,1..n]$ is computed, and $s[i,j]$ is the split position for $A_iA_{i+1}\ldots A_j$, i.e, $A_i\ldots A_{s[i,j]}$ and $A_{s[i,j]+1}\ldots A_j$, thus, the parenthesization order can be obtained from $s[1..n,1..n]$ recursively, beginning from $s[1,n]$.

MCM DP Steps

- Step 4, algorithm

PRINT-OPTIMAL-PARENS(s, i, j)

if $i = j$
  then print "A"$_i$
  else print "("
    PRINT-OPTIMAL-PARENS$(s, i, s[i, j])$
    PRINT-OPTIMAL-PARENS$(s, s[i, j] + 1, j)$
    print ")"